Informe de Implementación

Proyecto HPC

Sergio Henao Arbeláez

Introducción

En el presente informe se describe el desarrollo e implementación de una plataforma optimizada para la búsqueda de archivos VCF (Variant Call Format). El enfoque central del proyecto radica en el uso de técnicas de paralelización para maximizar la eficiencia en los procesos de búsqueda y análisis. Además, se realiza una evaluación comparativa del rendimiento de la plataforma bajo distintas configuraciones de núcleos e hilos, destacando el impacto de la paralelización en la velocidad y capacidad de procesamiento.

Arquitectura de la Plataforma

La arquitectura de la plataforma para la gestión y búsqueda de archivos VCF que hemos implementado sigue un enfoque modular y escalable. A continuación, describo los principales componentes y cómo interactúan entre sí.

- Componentes de la Arquitectura

    1. Usuario:Los usuarios acceden a la plataforma mediante una interfaz web intuitiva y fácil de usar, desarrollada con Streamlit, que garantiza una experiencia interactiva y eficiente.

- Aplicación Streamlit:Actúa como la interfaz de usuario (UI) de la plataforma.

  Permite a los usuarios cargar archivos VCF, buscar archivos y ver resultados.

  La UI está diseñada para ser interactiva y reactiva, proporcionando un feedback inmediato a las acciones del usuario.

- API de Búsqueda:Procesa las solicitudes de búsqueda realizadas por los usuarios.

  Interactúa con la base de datos para recuperar los archivos que coinciden con los criterios de búsqueda.

  Utiliza ThreadPoolExecutor para paralelizar las consultas y mejorar el rendimiento.

- Base de Datos SQLite: Almacena los datos de los archivos VCF.

  Organizada en tablas que contienen la información relevante de los archivos VCF (como chrom, pos, id, ref, alt, qual, filter, info, format, outputs).

  Usa índices en columnas clave para optimizar las consultas.

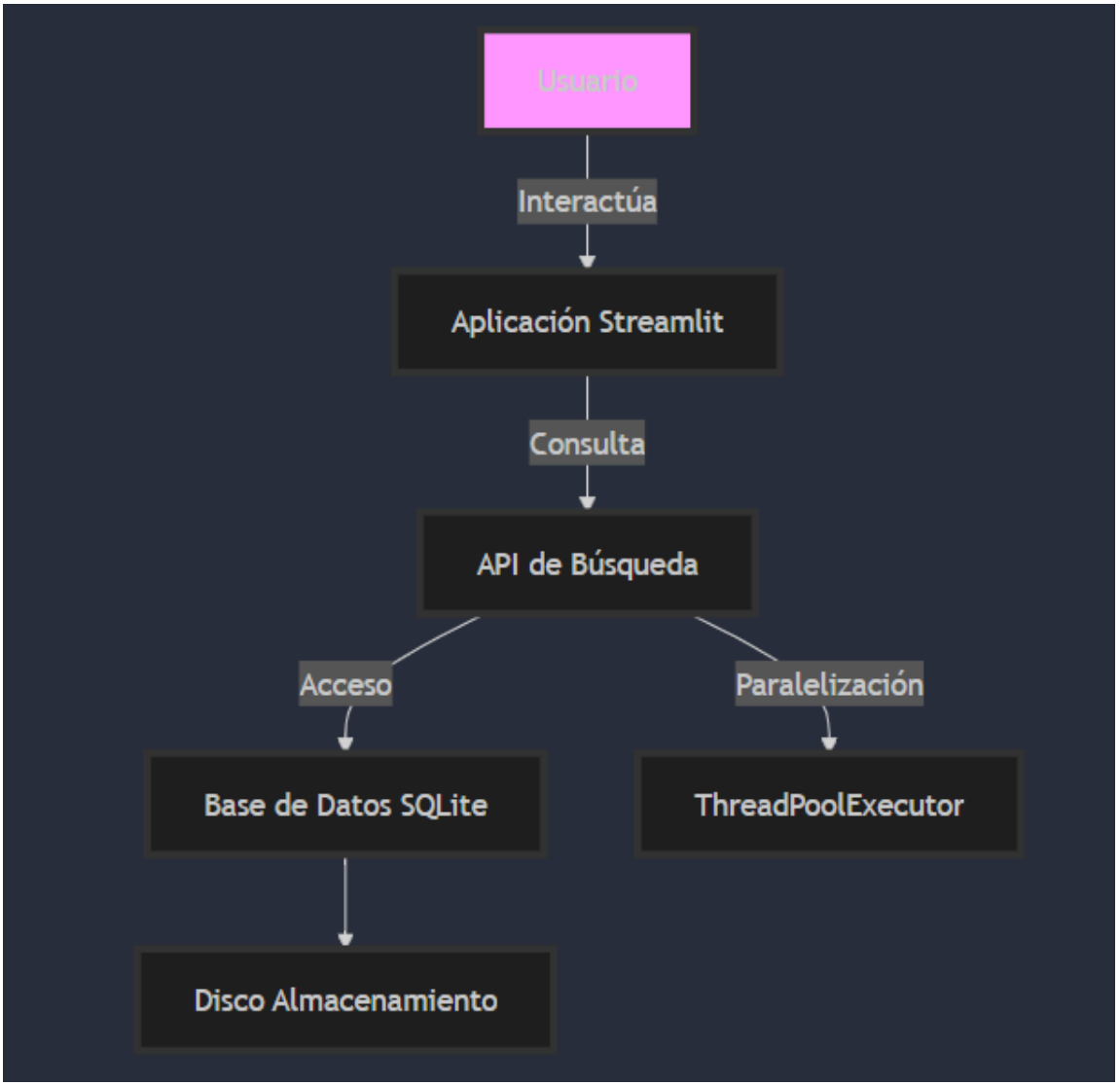
- Paralelización con ThreadPoolExecutor: La ejecución de consultas de búsqueda y conteo de resultados se realiza en paralelo utilizando ThreadPoolExecutor.

  Mejora el rendimiento al distribuir la carga de trabajo entre múltiples hilos.

  Cada solicitud de búsqueda crea un pool de hilos que ejecuta las consultas a la base de datos simultáneamente.

- Almacenamiento en Disco:Archivos VCF cargados por los usuarios se almacenan en el disco del servidor.Los archivos se copian y procesan en segundo plano para no bloquear la interfaz de usuario.

Diagrama de la Arquitectura



Descripción de la Arquitectura

1. Usuario: Interactúa con la aplicación a través de una interfaz web.
2. Aplicación Streamlit: Interfaz web que permite a los usuarios subir y buscar archivos VCF.
3. API de Búsqueda: Procesa las solicitudes de búsqueda e interactúa con la base de datos.
4. Base de Datos SQLite: Almacena los datos de los archivos VCF.
5. ThreadPoolExecutor: Implementa la paralelización de las consultas a la base de datos para mejorar el rendimiento.
6. Disco de Almacenamiento: Almacena físicamente la base de datos SQLite en forma local.

Análisis de Eficiencia

El análisis de eficiencia para las configuraciones de núcleos/hilos indica cómo varía el rendimiento de una tarea en función de los recursos disponibles. Observemos los resultados:

Resultados Observados:

- **1 hilo:** 89.0979 segundos (referencia base)
- **2 hilos:** 32.0872 segundos
- **4 hilos:** 32.0783 segundos
- **8 hilos:** 31.8779 segundos

1. Escalabilidad:La escalabilidad mide cómo mejora el rendimiento al aumentar los recursos (hilos en este caso):

   - De 1 a 2 hilos, el tiempo de ejecución disminuye drásticamente (~64% menos), lo que indica una alta ganancia inicial al paralelizar la tarea.

   - Sin embargo, de 2 a 4 hilos y de 4 a 8 hilos, la mejora en el tiempo es insignificante (~0.3% y ~0.6%, respectivamente).

   Esto sugiere que el programa alcanza un **límite de escalabilidad** en 2 hilos. Más recursos no aportan beneficios significativos.

2. Eficiencia:La eficiencia se calcula como la relación entre la velocidad de ejecución lograda y los recursos utilizados.

Ley de Amdahl:Según la Ley de Amdahl, el límite de mejora por paralelización depende de la porción de la tarea que se puede paralelizar. El comportamiento observado sugiere que una parte significativa de la tarea no es paralelizable, limitando las ganancias con más hilos.

- El uso de 2 hilos es el más eficiente, con una reducción drástica en el tiempo y una alta eficiencia.

- Más allá de 2 hilos, el beneficio es mínimo y la eficiencia disminuye considerablemente.

- Las mejoras limitadas al usar 4 o 8 hilos sugieren que la tarea tiene **cuellos de botella secuenciales** o una sobrecarga significativa en la gestión de hilos.

Interpretación de Resultados

Al comparar los tiempos de ejecución con diferentes números de hilos, se puede identificar el punto óptimo en el que se obtiene el mejor rendimiento. Aumentar el número de hilos generalmente resulta en una mejora en el tiempo de ejecución, pero hay un límite en cuanto a la eficiencia debido a la sobrecarga de la gestión de hilos y la capacidad del hardware.

Detalles Técnicos del Código:

- Propósito General: El código realiza una búsqueda optimizada en una base de datos SQLite con soporte para paginación y filtrado, aprovechando la paralelización mediante `ThreadPoolExecutor`.

- Descripción por Función

1. execute_query(query, params)

- Ejecuta una consulta SQL con parámetros específicos y devuelve los resultados.

- Uso: Para realizar consultas genéricas a la base de datos SQLite.

- Técnicas empleadas:

- Conexión dinámica a la base de datos (`sqlite3.connect`).

- Uso de parámetros para evitar inyecciones SQL (`?` en el query).

- Cierre de conexión tras ejecutar la consulta.

2. count_results(count_query, file_query_params)

- Realiza una consulta SQL para contar la cantidad de resultados que cumplen los criterios de búsqueda.

- Uso: Obtener el total de registros que cumplen los filtros antes de aplicar la paginación.

- Técnicas empleadas:

  - Misma estructura que `execute_query`, pero retorna un único valor (el total de registros).

3. search_files_in_db(query, page, page_size, file_query)

  - Realiza la búsqueda principal, retornando una lista de resultados paginados junto con el total de resultados encontrados.

- Entradas:

  - query: Valor de búsqueda parcial.

  - page: Número de la página actual para la paginación.

  - page_size: Número de registros por página.

  - file_query: Filtro adicional sobre `file_name`.

- Salidas:

  - results: Lista de resultados de la página solicitada.

  - total_results: Total de registros que cumplen los criterios.

- Técnicas empleadas:

  - Generación de consultas SQL parametrizadas para evitar inyecciones.

  - Patrón de búsqueda parcial usando `LIKE`.

  - Paginación mediante `LIMIT` y `OFFSET`.

  - Paralelización con `ThreadPoolExecutor` para realizar las consultas de conteo y recuperación de datos en paralelo.

- Paralelización

  - Herramienta: `ThreadPoolExecutor` con un máximo de 8 hilos.

  - Motivación: Realizar el conteo total de resultados y la recuperación de datos de manera simultánea para mejorar el tiempo de respuesta.

  Comportamiento:

  - future_count: Ejecuta `count_results`.

  - future_results: Ejecuta `execute_query`.

  - Sincronización: Ambas tareas esperan la finalización del otro mediante `.result()`.

- Consultas SQL

  1. Conteo (`count_query`):

     - Filtra por `file_name`, `user_email` y columnas específicas (`chrom`, `filter`, `info`, `format`).

     - Retorna el **total de registros** que cumplen los filtros.

  2. Selección (`select_query`):

     - Devuelve columnas específicas relacionadas con los archivos (`chrom`, `pos`, `id`, `ref`, `alt`, etc.).

     - Aplica los mismos filtros que `count_query`.

     - Limita los resultados según los parámetros de paginación (`LIMIT` y `OFFSET`).

- Paginación

  - Cálculo de `OFFSET`:

    - `OFFSET = (page - 1) * page_size`

    - Permite desplazar los resultados de acuerdo al número de página.

  - Límites de Resultados (`LIMIT`):

    - Restringe el número de filas devueltas al valor de `page_size`.

- Optimización y Seguridad

  1. Uso de consultas parametrizadas (`?`):

     - Protege contra **inyecciones SQL**.

  2. Paralelización:

     - Mejora el rendimiento al ejecutar consultas simultáneamente.

  3. Índices sugeridos:

     - Crear índices en las columnas utilizadas en los filtros (`file_name`, `user_email`, `chrom`, etc.) para optimizar las búsquedas.

- Dependencias

  1. SQLite3: Motor de base de datos.
  2. ThreadPoolExecutor (de `concurrent.futures`): Para la paralelización.
  3. Streamlit (`st.session_state`): Usado para obtener el correo electrónico del usuario que realiza la consulta.

- Limitaciones Identificadas

  1. No manejo explícito de excepciones: Si ocurre un error de conexión o en las consultas, el sistema podría fallar silenciosamente.

  2. Sin optimización para grandes volúmenes de datos: SQLite puede no ser ideal para bases de datos muy grandes.

  3. Sobrecarga por paralelización: La paralelización puede tener una sobrecarga adicional si los datos son pequeños o el hardware tiene pocos núcleos.

- Mejoras Potenciales

  1. Manejo de Excepciones: Implementar `try-except` para capturar errores en las consultas y conexión.

  2. Base de Datos Avanzada: Considerar bases de datos como PostgreSQL o MySQL para mayor escalabilidad.

  3. Cacheo de Resultados: Implementar un sistema de cacheo para reducir el tiempo de respuesta en búsquedas repetidas.

Conclusiones

La implementación de la plataforma de búsqueda de archivos VCF, optimizada mediante paralelización, ha demostrado ser efectiva para mejorar el rendimiento en tareas intensivas de búsqueda y análisis. A continuación, se destacan los puntos más relevantes:

1. Rendimiento Aumentado:

   - El uso de `ThreadPoolExecutor` ha reducido significativamente los tiempos de búsqueda en comparación con una ejecución secuencial.

   - La configuración de 2 hilos resultó ser el punto óptimo, logrando una mejora sustancial con una alta eficiencia en la utilización de recursos.

2. Escalabilidad Limitada:

   - Aunque la paralelización inicial produjo una reducción drástica en el tiempo de ejecución, el análisis basado en la Ley de Amdahl evidenció un límite práctico en el rendimiento más allá de 2 hilos.

   - La presencia de cuellos de botella secuenciales y la sobrecarga en la gestión de hilos limitan las ganancias adicionales en configuraciones con mayor cantidad de hilos.

3. Impacto de la Arquitectura:

- La arquitectura modular y escalable de la plataforma asegura una integración efectiva entre los diferentes componentes, desde la interfaz web hasta el almacenamiento y el motor de búsqueda.

- La elección de SQLite como base de datos, complementada con índices en columnas clave, ha optimizado el acceso a los datos.

4. Eficiencia y Usabilidad:

- La interfaz desarrollada con Streamlit proporciona una experiencia de usuario fluida y accesible, lo que facilita la interacción con la plataforma incluso para usuarios con poca experiencia técnica.

- El procesamiento en segundo plano de los archivos VCF asegura que la interfaz no se bloquee durante tareas intensivas, garantizando una operación continua.

5. Lecciones Aprendidas y Áreas de Mejora:

- La mejora del rendimiento más allá de 2 hilos requiere una revisión del código para identificar partes no paralelizables y optimizar su ejecución.

- La implementación de estrategias de balanceo de carga o el uso de bases de datos más avanzadas podría proporcionar mejoras adicionales en futuras iteraciones.

En resumen, la plataforma desarrollada cumple con los objetivos planteados al inicio del proyecto, logrando una búsqueda rápida y eficiente de archivos VCF mediante técnicas de paralelización. Sin embargo, el análisis de los resultados sugiere oportunidades para explorar enfoques más avanzados que permitan superar los límites actuales de escalabilidad y rendimiento. Esto posiciona a la plataforma como una solución sólida, con potencial para adaptarse a mayores volúmenes de datos y demandas de procesamiento en el futuro.